, , 1–25 ()

Strategies in Combined Learning via Logic Programs

EVELINA LAMMA, FABRIZIO RIGUZZI elamma,friguzzi@deis.unibo.it
DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy,

LUíS MONIZ PEREIRA lmp@di.fct.unl.pt
Centro de Inteligência Artificial (CENTRIA), Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2825 Monte da Caparica, Portugal

Editor:

Abstract. We discuss the adoption of a three-valued setting for inductive concept learning. Distinguishing between what is true, what is false and what is unknown can be useful in situations where decisions have to be taken on the basis of scarce, ambiguous, or downright contradictory information. In a three-valued setting, we learn a definition for both the target concept and its opposite, considering positive and negative examples as instances of two disjoint classes. To this purpose, we adopt, for the first time in a journal paper, Extended Logic Programs (ELP) under a Well-Founded Semantics with explicit negation (*WFSX*) as the representation formalism for learning, and show how ELPs can be used to specify combinations of strategies in a declarative way also coping with contradiction and exceptions.

Explicit negation is used to represent the opposite concept, while default negation is used to ensure consistency and to handle exceptions to general rules. Exceptions to a concept are identified from examples of the opposite training set.

Standard Inductive Logic Programming techniques are employed to learn the concept and its opposite. Depending on the adopted technique, we can learn the most general or the least general definition. Thus, four epistemological varieties occur, resulting from the combination of most general and least general solutions for the positive and negative concept. We discuss the factors that should be taken into account when choosing and strategically combining the generality levels for positive and negative concepts.

In the paper, we also handle the issue of strategic combination of possibly contradictory learnt definitions of a predicate and its explicit negation.

All in all we show that extended logic programs, under well-founded semantics with explicit negation add expressivity to learning tasks, and allow the tackling of a number of representation and strategic issues in a principled way.

Our techniques have been implemented and examples run on a state-of-the-art logic programming system with tabling which implements *WFSX*.

Keywords: Inductive Logic Programming, Non-monotonic Learning, Multi-strategy Learning, Explicit Negation, Contradiction Handling.

1. Introduction

Most work on inductive concept learning considers a two-valued setting. In such a setting, what is not entailed by the learned theory is considered false, on the basis of the Closed World Assumption (CWA) [44]. However, in practice, it is more often the case that we are confident about the truth or falsity of only a limited number of facts, and are not able to draw any conclusion about the remaining ones, because the available information is too scarce. Like it has been pointed out in [13, 10], this is typically the case of an autonomous agent that, in an incremental way, gathers information from its surrounding world. Such an agent needs to distinguish between what is true, what is false and what is unknown, and therefore needs to learn within a richer three-valued setting.

To this purpose, we adopt the class of *Extended Logic Programs* (ELP, for short, in the sequel) as the representation language for learning in a three-valued setting. ELP contains two kinds of negation: default negation plus a second form of negation, called *explicit*, whose combination has been recognized elsewhere as a very expressive means of knowledge representation. The adoption of ELP allows one to deal directly in the language with incomplete and contradictory knowledge, with exceptions through default negation, as well as with truly *negative* information by means of explicit negation [39, 2, 3]. For instance, in [2, 5, 16, 9, 32] it is shown how ELP is applicable to such diverse domains of knowledge representation as concept hierarchies, reasoning about actions, belief revision, counterfactuals, diagnosis, updates and debugging.

In this work, we show, for the first time in a journal, that various approaches and strategies can be adopted in Inductive Logic Programming (ILP, henceforth) for learning with ELP under an extension of well-founded semantics. As in [25, 24], where answer-sets semantics is used, the learning process starts from a set of positive and negative examples plus some background knowledge in the form of an extended logic program. Positive and negative information in the training set are treated equally, by learning a definition for both a positive concept p and its (explicitly) negated concept $\neg p$. Coverage of examples is tested by adopting the SLX interpreter for ELP under the Well-Founded Semantics with explicit negation ($WFSX$) defined in [2, 16], and valid for its paraconsistent version [9].

Default negation is used in the learning process to handle *exceptions* to general rules. Exceptions to a positive concept are identified from negative examples, whereas exceptions to a negative concept are identified from positive examples.

In this work, we adopt standard ILP techniques to learn some concept and its opposite. Depending on the technique adopted, one can learn the most general or the least general definition for each concept. Accordingly, four epistemological varieties occur, resulting from the mutual combination of most general and least general solutions for the positive and negative concept. These possibilities are expressed via ELP, and we discuss some of the factors that should be taken into account when choosing the level of generality of each, and their combination, to define a specific learning strategy, and how to cope with contradictions. (In the paper, we concentrate on single predicate learning, for the sake of simplicity.)

Indeed, separately learned positive and negative concepts may conflict and, in order to handle possible *contradiction*, contradictory learned rules are made defeasible by making the learned definition for a positive concept p depend on the default negation of the negative concept $\neg p$, and vice-versa. I.e., each definition is introduced as an exception to the other. This way of coping with contradiction can be even generalized for learning n disjoint classes, or modified in order to take into account preferences among multiple learning agents or information sources (see [28]).

The paper is organized as follows. We first motivate the use of ELP as target and background language in section 2, and introduce the new ILP framework in section 3. We then examine, in section 4, factors to be taken into account when choosing the level of generality of learned theories. Section 5 proposes how to combine the learned definitions within ELP in order to avoid inconsistencies on unseen atoms and their opposites, through the use of mutually defeating ("non-deterministic") rules, and how to incorporate exceptions through negation by default. A description of our algorithm for learning ELP follows next, in section 6, and the overall system implementation in section 7. Section 8 evaluates the obtained classification accuracy. Finally, we examine related works in section 9, and conclude.

2. Logic Programming and Epistemic Preliminaries

In this section, we first discuss the motivation for three-valuedness and two types of negation in knowledge representation and provide basic notions of extended logic programs and WFSX.

2.1. Three-valuedness, default and explicit negation

Artificial Intelligence (AI) needs to deal with knowledge in flux, and less than perfect conditions, by means of more dynamic forms of logic than classical logic. Much of this has been the focus of research in Logic Programming (LP), a field of AI which uses logic directly as a programming language[1], and provides specific implementation methods and efficient working systems to do so [2].

Horn clause notation is used to express that conclusions must be supported, "caused", by some premises. Implication is unidirectional, i.e., not contrapositive: "causes" do not run backwards.

Various extensions of LP have been introduced to cope with knowledge representation issues. For instance, default negation of an atom P, "*not P*", was introduced by AI to deal with lack of information, a common situation in the real world. It introduces non-monotonicity into knowledge representation. Indeed, conclusions might not be solid because the rules leading to them may be defeasible. For instance, we don't normally have explicit information about who is or is not the lover of whom, though that kind of information may arrive unexpectedly. Thus we write[3]:

$$faithful(H,K) \leftarrow married(H,K), not\ lover(H,L)$$

I.e., if we have no evidence to conclude $lover(H,L)$ for some L given H, we can assume it false for all L given H.

Mark that *not* should grant positive and negative information equal standing. That is, we should equally be able to write:

$$\neg faithful(H,K) \leftarrow married(H,K), not\ \neg lover(H,L)$$

to model instead a world where people are unfaithful by default or custom, and where it is required to explicitly prove that someone does not take any lover before concluding that person not unfaithful.

Since information is normally expressed positively, by dint of mental and linguistic economics, through Closed World Assumption (CWA), the absent, non explicitly obtainable information, is usually the negation of positive information. Which means, when no information is available about lovers, that $\neg lover(H, L)$ is true by CWA, whereas $lover(H, L)$ is not. Indeed, whereas the CWA is indispensable in some contexts, viz. at airports flights not listed are assumed non-existent, in others that is not so: though one's residence might not be listed in the phone book, it may not be ruled out that it exists and has a phone.

These epistemologic requisites can be reconciled by reading '$\neg$' above not as classical negation, which complies with the excluded middle provision, but as yet a new form of negation, dubbed in Logic Programming "explicit negation" [39] (which ignores that provision), and adopted in ELP.

This requires the need for revising assumptions and for introducing a third truth-value, named "undefined", into the framework. In fact, when we combine, for instance, the viewpoints of the two above worlds about faithfulness we become confused: assuming $married(H, K)$ for some H and K, it now appears that both $faithful(H, K)$ and $\neg faithful(H, K)$ are contradictorily true. Indeed, since we have no evidence for $lover(H, L)$ nor $\neg lover(H, L)$ because there simply is no information about them, we make two simultaneous assumptions about their falsity. But when any assumption leads to contradiction one should retract it, which in a three-valued setting means making it *undefined*.

The imposition of undefinedness for $lover(H, L)$ and $\neg lover(H, L)$ can be achieved simply, by adding to our knowledge the clauses:

$$\begin{aligned} \neg lover(H, L) &\leftarrow not\ lover(H, L) \\ lover(H, L) &\leftarrow not\ \neg lover(H, L) \end{aligned}$$

thereby making $faithful(H, K)$ and $\neg faithful(H, K)$ undefined too. Given no other information, we can now prove neither of $lover(H, L)$ nor $\neg lover(H, L)$ true, or false. Any attempt to do it runs into a self-referential circle involving default negation, and so the safest, skeptical, third option is to take no side in this marital dispute, and abstain from believing either.

Even in presence of self-referential loops involving default negations, the well-founded semantics of logic programs (WFS) assigns to the literals in the above two clauses the truth value *undefined*, in its knowledge skeptical well-founded model, but allows also for the other two, incompatible non truth-minimal, more credulous models.

2.2. Extended Logic Programs

An *extended logic program* is a finite set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_n$$

with $n \geq 0$, where L_0 is an objective literal, $L_1, \ldots, L_n$ are literals and each rule stands for the sets of its ground instances. *Objective literals* are of the form A or $\neg A$, where A is an atom, while a *literal* is either an objective literal L or its default negation $not\ L$. $\neg A$ is said the *opposite* literal of A (and vice versa), where $\neg\neg A = A$, and $not\ A$ the *complementary* literal of A (and vice versa). By $not\ \{A_1, \ldots, A_n\}$ we mean $\{not\ A_1, \ldots, not\ A_n\}$ where A_is are literals. By $\neg\ \{A_1, \ldots, A_n\}$ we mean $\{\neg A_1, \ldots, \neg A_n\}$. The set of all objective literals of a program P is called its *extended Herbrand base* and is represented as $H^E(P)$. An *interpretation* I of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $H^E(P)$. Objective literals in T are said to be *true* in I, objective literals in F are said to be *false* in I and those in $H^E(P) - I$ are said to be *undefined* in I. We introduce in the language the proposition $\mathbf{u}$ that is undefined in every interpretation I.

WFSX extends the well founded semantics (*WFS*) [46] for normal logic programs to the case of extended logic programs. *WFSX* is obtained from *WFS* by adding the coherence principle relating the two forms of negation: "*if L is an objective literal and $\neg L$ belongs to the model of a program, then also not L belongs to the model*", i.e., $\neg L \rightarrow not\ L$.

Notice that, thanks to this principle, any interpretation $I = T \cup not\ F$ of an extended logic program P considered by *WFSX* semantics is non-contradictory, i.e., there is no pair of objective literals A and $\neg A$ of program P such that A belongs to T and $\neg A$ belongs to T [2]. The definition of *WFSX* is reported in Appendix. If an objective literal A is true in the *WFSX* of an ELP P we write $P \models_{WFSX} A$.

Let us now show an example of *WFSX* in the case of a simple program.

EXAMPLE: Consider the following extended logic program:

$$\begin{array}{lllll} \neg a & \leftarrow & . & \qquad\qquad & b \leftarrow not\ b. \\ a & \leftarrow & b. & & \end{array}$$

A *WFSX* model of this program is $M = \{\neg a, not\ \neg b, not\ a\}$: $\neg a$ is true, a is false, $\neg b$ is false (there are no rules for $\neg b$) and b is undefined. Notice that $not\ a$ is in the model since it is implied by $\neg a$ via the coherence principle. □

One of the most important characteristic of *WFSX* is that it provides a semantics for an important class of extended logic programs: the set of *non-stratified* programs, i.e., the set of programs that contain recursion through default negation. An extended logic program is *stratified* if its *dependency graph* does not contain any cycle with an arc labelled with $-$. The *dependency graph* of a program P is a labelled graph with a node for each predicate of P and an arc from a predicate p to a predicate q if q appears in the body of clauses with p in the head. The arc is labelled with $+$ if q appears in an objective literal in the body and with $-$ if it appears in a default literal.

Non-stratified programs are very useful for knowledge representation because the *WFSX* semantics assigns the truth value undefined to the literals involved in the recursive cycle through negation, as shown in section 2.1 for $lover(H, L)$

and $\neg lover(H, L)$. In section 5 we will employ non stratified programs in order to resolve possible contradictions.

WFSX was chosen among the other semantics for extended logic programs, answer-sets [21] and three-valued strong negation [3], because none of the others enjoy the property of relevance [2, 3] for non-stratified programs, i.e., they cannot have top-down querying procedures for non-stratified programs. Instead, for *WFSX* there exists a top-down proof procedure *SLX* [2], which is correct with respect to the semantics.

Cumulativity is also enjoyed by *WFSX*, i.e., if you add a lemma then the semantics does not change (see [2]). This property is important for speeding-up the implementation. By memorizing intermediate lemmas through tabling, the implementation of *SLX* greatly improves. Answer-set semantics, however, is not cumulative for non-stratified programs and thus cannot use tabling.

The *SLX* top-down procedure for *WFSX* relies on two independent kinds of derivations: T-derivations, proving truth, and TU-derivations proving non-falsity, i.e., truth or undefinedness. Shifting from one to the other is required for proving a default literal *not* L: the T-derivation of *not* L succeeds if the TU-derivation of L fails; the TU-derivation of *not* L succeeds if the T-derivation of L fails. Moreover, the T-derivation of *not* L also succeeds if the T-derivation of $\neg L$ succeeds, and the TU-derivation of L fails if the T-derivation of $\neg L$ succeeds (thus taking into account the *coherence principle*). Given a goal G that is a conjunction of literals, if G can be derived by *SLX* from an ELP P, we write $P \vdash_{SLX} G$

The *SLX* procedure is amenable to a simple pre-processing implementation, by mapping *WFSX* programs into *WFS* programs through the T-TU transformation [8]. This transformation is linear and essentially doubles the number of program clauses. Then, the transformed program can be executed in *XSB*, an efficient logic programming system which implements (with polynomial complexity) the *WFS* with tabling, and subsumes Prolog. Tabling in *XSB* consists in memoizing intermediate lemmas, and in properly dealing with non-stratification according to *WFS*. Tabling is important in learning, where computations are often repeated for testing the coverage or otherwise of examples, and allows computing the *WFS* with simple polynomial complexity on program size.

3. Learning in a Three-valued Setting

In real-world problems, complete information about the world is impossible to achieve and it is necessary to reason and act on the basis of the available partial information. In situations of incomplete knowledge, it is important to distinguish between what is true, what is false, and what is unknown or undefined.

Such a situation occurs, for example, when an agent incrementally gathers information from the surrounding world and has to select its own actions on the basis of such acquired knowledge. If the agent learns in a two-valued setting, it can encounter the problems that have been highlighted in [13]. When learning in a specific to general way, it will learn a cautious definition for the target concept and it will not be able to distinguish what is false from what is not yet known (see figure 1a). Supposing the target predicate represents the allowed actions, then the agent will not distinguish forbidden actions from actions with an outcome and this can restrict the agent acting power. If the agent learns in a general to specific way, instead, it will not know the difference between what is true and what is unknown (figure 1b) and, therefore, it can try actions with an unknown outcome. Rather, by learning in a three-valued setting, it will be able to distinguish between allowed actions, forbidden actions, and actions with an unknown outcome (figure 1c). In this way, the agent will know which part of the domain needs to be further explored and will not try actions with an unknown outcome unless it is trying to expand its knowledge.

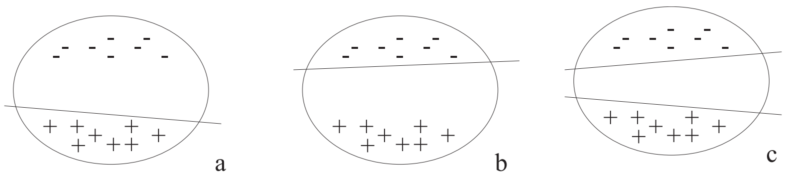

Figure 1. (taken from [13])(a,b): two-valued setting, (c): three-valued setting

We therefore consider a new learning problem where we want to learn an ELP from a background knowledge that is itself an ELP and from a set of positive and a set of negative examples in the form of ground facts for the target predicates. A learning problem for ELP's was first introduced in [25] where the notion of coverage was defined by means of truth in the answer-set semantics. Here the problem definition is modified to consider coverage as truth in the preferred *WFSX* semantics

Definition 1. [Learning Extended Logic Programs]
Given:

- a set $\mathcal{P}$ of possible (extended logic) programs
- a set E^+ of positive examples (ground facts)
- a set E^- of negative examples (ground facts)
- a non-contradictory extended logic program B (*background knowledge*[4])

Find:

- an extended logic program $P \in \mathcal{P}$ such that
  - $\forall e \in E^+ \cup \neg E^-$, $B \cup P \models_{WFSX} e$ (*completeness*)
  - $\forall e \in \neg E^+ \cup E^-$, $B \cup P \not\models_{WFSX} e$ (*consistency*)

  where $\neg E = \{\neg e | e \in E\}$.

We suppose that the training sets E^+ and E^- are disjoint. However, the system is also able to work with overlapping training sets.

The learned theory will contain rules of the form:

$$\begin{aligned} p(\vec{X}) &\leftarrow Body^+(\vec{X}) \\ \neg p(\vec{X}) &\leftarrow Body^-(\vec{X}) \end{aligned}$$

for every target predicate p, where $\vec{X}$ stands for a tuple of arguments. In order to satisfy the completeness requirement, the rules for p will entail all positive examples while the rules for $\neg p$ will entail all (explicitly negated) negative examples. The consistency requirement is satisfied by ensuring that both sets of rules do not entail instances of the opposite element in either of the training sets.

Note that, in the case of extended logic programs, the consistency with respect to the training set is equivalent to the requirement that the program is non-contradictory on the examples. This requirement is enlarged to require that the program be non-contradictory also for unseen atoms, i.e., $B \cup P \not\models L \wedge \neg L$ for every atom L of the target predicates.

We say that an example e is *covered* by program P if $P \models_{WFSX} e$. Since the *SLX* procedure is correct with respect to *WFSX*, even for contradictory programs, coverage of examples is tested by verifying whether $P \vdash_{SLX} e$.

Our approach to learning with extended logic programs consists in initially applying conventional ILP techniques to learn a positive definition from E^+ and E^- and a negative definition from E^- and E^+. In these techniques, the *SLX* procedure substitutes the standard Logic Programming proof procedure to test the coverage of examples.

The ILP techniques to be used depend on the level of generality that we want to have for the two definitions: we can look for the Least General Solution (LGS) or the Most General Solution (MGS) of the problem of learning each concept and its complement. In practice, LGS and MGS are not unique and real systems usually learn theories that are not the least nor most general, but closely approximate one of the two. In the following, these concepts will be used to signify approximations to the theoretical concepts.

LGSs can be found by adopting one of the bottom-up methods such as relative least general generalization ($rlgg$) [40] and the GOLEM system [37], inverse resolution [36] or inverse entailment [30]. Conversely, MGSs can be found by adopting a top-down refining method (cf. [31]) and a system such as FOIL [43] or Progol [35].

4. Strategies for Combining Different Generalizations

The generality of concepts to be learned is an important issue when learning in a three-valued setting. In a two-valued setting, once the generality of the definition is chosen, the extension (i.e., the generality) of the set of false atoms is undesirably and automatically decided, because it is the complement of the true atoms set. In a three-valued setting, rather, the extension of the set of false atoms depends on the generality of the definition learned for the negative concept. Therefore, the corresponding level of generality may be chosen independently for the two definitions, thus affording four epistemological cases. The adoption of ELP allows to express case combination in a declarative and smooth way.

Furthermore, the generality of the solutions learned for the positive and negative concepts clearly influences the interaction between the definitions. If we learn the MGS for both a concept and its opposite, the probability that their intersection is non-empty is higher than if we learn the LGS for both. Accordingly, the decision as to which type of solution to learn should take into account the possibility of interaction as well: if we want to reduce this possibility, we have to learn two LGS, if we do not care about interaction, we can learn two MGS. In general, we may learn different generalizations and combine them in distinct ways for different strategic purposes within the same application problem.

The choice of the level of generality should be made on the basis of available knowledge about the domain. Two of the criteria that can be taken into account are the damage or risk that may arise from an erroneous classification of an unseen object, and the confidence we have in the training set as to its correctness and representativeness.

When classifying an as yet unseen object as belonging to a concept, we may later discover that the object belongs to the opposite concept. The more we generalize a concept, the higher is the number of unseen atoms covered by the definition and the higher is the risk of an erroneous classification. Depending on the damage that may derive from such a mistake, we may decide to take a more cautious or a more confident approach. If the possible damage from an over extensive concept is high, then one should learn the LGS for that concept, if the possible damage is low then one can generalize the most and learn the MGS. The overall risk will depend too on the use of the learned concepts within other rules.

The problem of selecting a solution of an inductive problem according to the cost of misclassifying examples has been studied in a number of works. PREDICTOR [22] is able to select the cautiousness of its learning operators by means of metaheuristics. These metaheuristics make the selection based on a user-input penalty for prediction error. [41] provides a method to select classifiers given the cost of misclassifications and the prior distribution of positive and negative instances. The method is based on the Receiver Operating Characteristic (ROC) graph from signal theory that depicts classifiers as points in a graph with the number of false positive on the X axis and the number of true positive on the Y axis. In [38] it is discussed how the different costs of misclassifying examples can be taken into account into a number of algorithms: decision tree learners, Bayesian classifiers and decision lists learners. The Reduced Cost Algorithm is presented that selects and order rules after they have been learned in order to minimize misclassification costs. Moreover, an algorithm for pruning decision lists is presented that attempts to minimize costs while avoiding overfitting. In [23] it is discussed how the penalty incurred if a learner outputs the wrong classification is considered in order to decide whether to acquire additional information in an active learner.

As regards the confidence in the training set, we can prefer to learn the MGS for a concept if we are confident that examples for the opposite concept are correct and representative of the concept. In fact, in top-down methods, negative examples are used in order to delimit the generality of the solution. Otherwise, if we think that examples for the opposite concept are not reliable, then we should learn the LGS.

In the following, we present a realistic example of the kind of reasoning that can be used to choose and specify the preferred level of generality, and discuss how to strategically combine the different levels by employing ELP tools to learning.

EXAMPLE: Consider a person living in a bad neighbourhood in Los Angeles. He is an honest man and to survive he needs two concepts, one about who is likely to attack him, on the basis of appearance, gang membership, age, past dealings, etc. Since he wants to take a cautious approach, he maximizes *attacker* and minimizes $\neg attacker$, so that his $attacker1$ concept allows him to avoid dangerous situations.

$attacker1(X) \leftarrow attacker_{MGS}(X)$

$\neg attacker1(X) \leftarrow \neg attacker_{LGS}(X)$

Another concept he needs is the type of beggars he should give money to (he is a good man) that actually seem to deserve it, on the basis of appearance, health, age, etc. Since he is not rich and does not like to be tricked, he learns a $beggar1$ concept by minimizing *beggar* and maximizing $\neg beggar$, so that his beggar concept allows him to give money strictly to those appearing to need it without faking.

$beggar1(X) \leftarrow beggar_{LGS}(X)$

$\neg beggar1(X) \leftarrow \neg beggar_{MGS}(X)$

However rejected beggars, especially malicious ones, may turn into attackers, in this very bad neighbourhood. Consequently, if he thinks a beggar might attack him he had better be more permissive about who is a beggar and placate him with money. In other words, he should maximize *beggar* and minimize $\neg beggar$ in a $beggar2$ concept.

$beggar2(X) \leftarrow beggar_{MGS}(X)$

$\neg beggar2(X) \leftarrow \neg beggar_{LGS}(X)$

These concepts can be used in order to minimize his risk taking when he carries, by his standards, a lot of money and meets someone who is likely to be an attacker, with the following kind of reasoning:

$run(X) \leftarrow lot\_of\_money(X), meets(X,Y), attacker1(Y), not\ beggar2(Y)$

$\neg run(X) \leftarrow lot\_of\_money(X), give\_money(X,Y)$

$give\_money(X,Y) \leftarrow meets(X,Y), beggar1(Y)$

$give\_money(X,Y) \leftarrow meets(X,Y), attacker1(Y), beggar2(Y)$

If he does not have a lot of money on him, he may prefer not to run as he risks being beaten up. In this case he has to relax his attacker concept into $attacker2$, but not relax it so much that he would use $\neg attacker_{MGS}$.

$\neg run(X) \leftarrow little\_money(X), meets(X,Y), attacker2(Y)$

$attacker2(X) \leftarrow attacker_{LGS}(X)$

$\neg attacker2(X) \leftarrow \neg attacker_{LGS}(X)$

The various notions of *attacker* and *beggar* are then learnt on the basis of previous experience the man has had (see [29]). □

5. Strategies for Eliminating Learned Contradictions

The learnt definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the objective literals in the intersection. In order to resolve the conflict, we must distinguish two types of literals in the intersection: those that belong to the training set and those that do not, also dubbed *unseen* atoms (see figure 2).

In the following, we discuss how to resolve the conflict in the case of unseen literals and of literals in the training set. We first consider the case in which the training sets are disjoint, and we later extend the scope to the case where there is a non-empty intersection of the training sets, when they are less than perfect. From now onwards, $\vec{X}$ stands for a tuple of arguments.

For unseen literals, the conflict is resolved by classifying them as undefined, since the arguments supporting the two classifications are equally strong. Instead, for literals in the training set, the conflict is resolved by giving priority to the classification stipulated by the training set. In other words, literals in a training set that are covered by the opposite definition are made as *exceptions* to that definition.

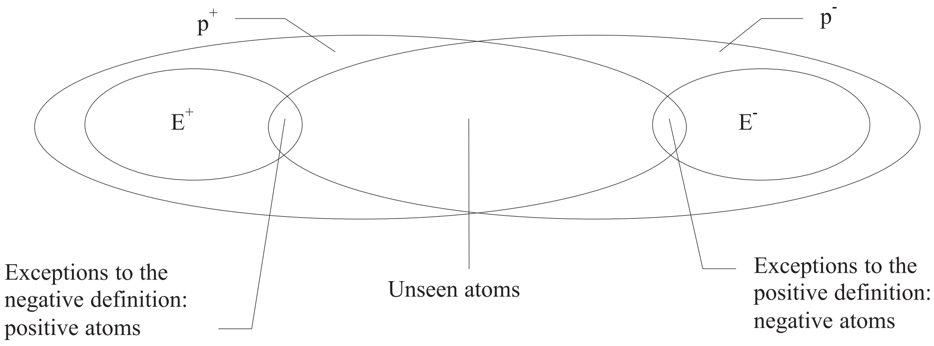

Figure 2. Interaction of the positive and negative definitions on exceptions.

Contradiction on Unseen Literals For unseen literals in the intersection, the undefined classification is obtained by making opposite rules mutually defeasible, or “non-deterministic” (see [5, 2]). The target theory is consequently expressed in the following way:

$$
\begin{aligned}
p(\vec{X}) &\leftarrow p^+(\vec{X}), not\ \neg p(\vec{X}) \\
\neg p(\vec{X}) &\leftarrow p^-(\vec{X}), not\ p(\vec{X})
\end{aligned}
$$

where $p^+(\vec{X})$ and $p^-(\vec{X})$ are, respectively, the definitions learned for the positive and the negative concept, obtained by renaming the positive predicate by p^+ and its explicit negation by p^-. From now onwards, we will indicate with these superscripts the definitions learned separately for the positive and negative concepts.

We want $p(\vec{X})$ and $\neg p(\vec{X})$ each to act as an exception to the other. In case of contradiction, this will introduce mutual circularity, and hence undefinedness according to *WFSX*. For each literal in the intersection of p^+ and p^-, there are two stable models, one containing the literal, the other containing the opposite literal. According to *WFSX*, there is a third (partial) stable model where both literals are undefined, i.e., no literal $p(\vec{X})$, $\neg p(\vec{X})$, *not* $p(\vec{X})$ or *not* $\neg p(\vec{X})$ belongs to the well-founded (or least partial stable) model. The resulting program contains a recursion through negation (i.e., it is non-stratified) but the top-down *SLX* procedure does not go into a loop because it comprises mechanisms for loop detection and treatment, which are implemented by XSB through tabling.

EXAMPLE: Let us consider the Example of section 4. In order to avoid contradictions on unseen atoms, the learned definitions must be:

$$\begin{array}{lcl}
attacker1(X) & \leftarrow & attacker^+_{MGS}(X), not\ \neg attacker1(X) \\
\neg attacker1(X) & \leftarrow & attacker^-_{LGS}(X), not\ attacker1(X) \\
beggar1(X) & \leftarrow & beggar^+_{LGS}(X), not\ \neg beggar1(X) \\
\neg beggar1(X) & \leftarrow & beggar^-_{MGS}(X), not\ beggar1(X) \\
beggar2(X) & \leftarrow & beggar^+_{MGS}(X), not\ \neg beggar2(X) \\
\neg beggar2(X) & \leftarrow & beggar^-_{LGS}(X), not\ beggar2(X) \\
attacker2(X) & \leftarrow & attacker^+_{LGS}(X), not\ \neg attacker2(X) \\
\neg attacker2(X) & \leftarrow & attacker^-_{LGS}(X), not\ attacker2(X)
\end{array}$$

□

Note that $p^+(\vec{X})$ and $p^-(\vec{X})$ can display as well the undefined truth value, either because the original background is non-stratified or because they rely on some definition learned for another target predicate, which is of the form above and therefore non-stratified. In this case, three-valued semantics can produce literals with the value "undefined", and one or both of $p^+(\vec{X})$ and $p^-(\vec{X})$ may be undefined. If one is undefined and the other is true, then the rules above make both p and $\neg p$ undefined, since the negation by default of an undefined literal is still undefined. However, this is counter-intuitive: a defined value should prevail over an undefined one.

In order to handle this case, we suppose that a system predicate $undefined(X)$ is available[5], that succeeds if and only if the literal X is undefined. So we add the following two rules to the definitions for p and $\neg p$:

$$\begin{array}{rcl}
p(\vec{X}) & \leftarrow & p^+(\vec{X}), undefined(p^-(\vec{X})) \\
\neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), undefined(p^+(\vec{X}))
\end{array}$$

According to these clauses, $p(\vec{X})$ is true when $p^+(\vec{X})$ is true and $p^-(\vec{X})$ is undefined, and conversely.

Contradiction on Examples Theories are tested for consistency on all the literals of the training set, so we should not have a conflict on them. However, in some cases, it is useful to relax the consistency requirement and learn clauses that cover a small amount of counter examples. This is advantageous when it would be otherwise impossible to learn a definition for the concept, because no clause is contained in the language bias that is consistent, or when an overspecific definition would be learned, composed of very many specific clauses instead of a few general ones. In such cases, the definitions of the positive and negative concepts may cover examples of the opposite training set. These must then be considered exceptions, which are then are due to abnormalities in the opposite concept.

Let us start with the case where some literals covered by a definition belong to the opposite training set. We want of course to classify these according to the classification given by the training set, by making such literals *exceptions*. To handle exceptions to classification rules, we add a negative default literal of the form *not* $abnorm_p(\vec{X})$ (resp. *not* $abnorm_{\neg p}(\vec{X})$) to the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$), to express possible abnormalities arising from exceptions. Then, for every exception $p(\vec{t})$, an individual fact of the form $abnorm_p(\vec{t})$ (resp. $abnorm_{\neg p}(\vec{t})$) is asserted so that the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$) does not cover the exception, while the opposite definition still covers it. In this way, exceptions will figure in the model of the theory with the correct truth value. The learned theory thus takes the form:

$$p(\vec{X}) \leftarrow p^+(\vec{X}), not\ abnorm_p(\vec{X}), not\ \neg p(\vec{X}) \quad (1)$$

$$\neg p(\vec{X}) \leftarrow p^-(\vec{X}), not\ abnorm_{\neg p}(\vec{X}), not\ p(\vec{X}) \quad (2)$$

$$p(\vec{X}) \leftarrow p^+(\vec{X}), undefined(p^-(\vec{X})) \quad (3)$$

$$\neg p(\vec{X}) \leftarrow p^-(\vec{X}), undefined(p^+(\vec{X})) \quad (4)$$

Abnormality literals have not been added to the rules for the undefined case because a literal which is an exception is also an example, and so must be covered by its respective definition; therefore it cannot be undefined.

Notice that if E^+ and E^- overlap for some example $p(\vec{t})$, then $p(\vec{t})$ is classified $false$ by the learned theory. A different behaviour would obtain by slightly changing the form of learned rules in order to adopt, for atoms of the training set, one classification as default and thus give preference to false (negative training set) or true (positive training set)

Individual facts of the form $abnorm_p(\vec{X})$ might then used as examples for learning a definition for $abnorm_p$ and $abnorm_{\neg p}$, as in [25, 19]. In turn, exceptions to the definitions of $abnorm_p$ and $abnorm_{\neg p}$ might be found and so on, thus leading to a hierarchy of exceptions (for our hierarchical learning of exceptions, see [27]).

EXAMPLE: Consider a domain containing entities a, b, c, d, e, f and suppose the target concept is $flies$. Let the background knowledge be:

$bird(a)$	$has\_wings(a)$	
$jet(b)$	$has\_wings(b)$	
$angel(c)$	$has\_wings(c)$	$has\_limbs(c)$
$penguin(d)$	$has\_wings(d)$	$has\_limbs(d)$
$dog(e)$		$has\_limbs(e)$
$cat(f)$		$has\_limbs(f)$

and let the training set be:

$E^+ = \{flies(a)\} \qquad\qquad E^- = \{flies(d), flies(e)\}$

A possible learned theory is:

$$\begin{array}{rcl} flies(X) & \leftarrow & flies^+(X), not\ abnormal_{flies}(X), not\ \neg flies(X) \\ \neg flies(X) & \leftarrow & flies^-(X), not\ flies(X) \\ flies(X) & \leftarrow & flies^+(X), undefined(flies^-(X)) \\ \neg flies(X) & \leftarrow & flies^-(X), undefined(flies^+(X)) \\ abnormal_{flies}(d) & & \end{array}$$

where $flies^+(X) \leftarrow has\_wings(X)$ and $flies(X)^- \leftarrow has\_limbs(X)$. □

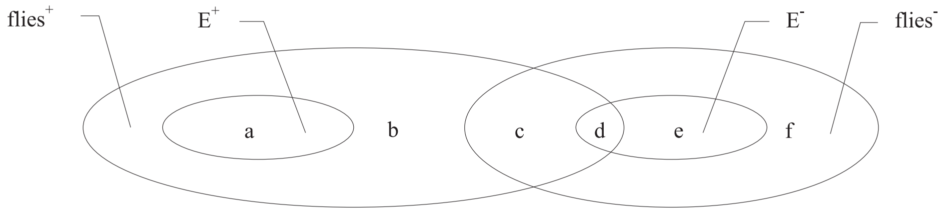

Figure 3. Coverage of definitions for opposite concepts

The example above and figure 3 show all the various cases for a literal when learning in a three-valued setting. a and e are examples that are consistently covered by the definitions. b and f are unseen literals on which there is no contradiction. c and d are literals where there is contradiction, but c is classified as undefined whereas d is considered as an exception to the positive definition and is classified as negative.

Identifying contradictions on unseen literals is useful in interactive theory revision, where the system can ask an oracle to classify the literal(s) leading to contradiction, and accordingly revise the least or most general solutions for p and for $\neg p$ using a theory revision system such as REVISE [7] or CLINT [12, 14]. Detecting uncovered literals points to theory extension.

Extended logic programs can be used as well to represent n disjoint classes $p_1, \ldots, p_n$. When one has to learn n disjoint classes, the training set contains a number of facts for a number of predicates $p_1, \ldots, p_n$. Let p_i^+ be a definition learned by using, as positive examples, the literals in the training set classified as belonging to p_i and, as negative examples, all the literals for the other classes. Then the following rules ensure consistency on unseen literals and on exceptions:

$$\begin{array}{rcl} p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), not\ abnormal_{p_1}(\vec{X}), not\ p_2(\vec{X}), \ldots, not\ p_n(\vec{X}) \\ p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), not\ abnormal_{p_2}(\vec{X}), not\ p_1(\vec{X}), not\ p_3(\vec{X}), \ldots, not\ p_n(\vec{X}) \\ \cdots & \leftarrow & \cdots \\ p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), not\ abnormal_{p_n}(\vec{X}), not\ p_1(\vec{X}), \ldots, not\ p_{n-1}(\vec{X}) \end{array}$$

$$
\begin{array}{lcl}
p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), undefined(p_2^+(\vec{X})), \ldots, undefined(p_n^+(\vec{X})) \\
p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), undefined(p_1^+(\vec{X})), undefined(p_3^+(\vec{X})), \ldots, undefined(p_n^+(\vec{X})) \\
\cdots & \leftarrow & \cdots \\
p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), undefined(p_1^+(\vec{X})), \ldots, undefined(p_{n-1}^+(\vec{X}))
\end{array}
$$

regardless of the algorithm used for learning the p_i^+.

6. An Algorithm for Learning Extended Logic Programs

The algorithm LIVE (Learning In a 3-Valued Environment) learns ELPs containing non-deterministic rules for a concept and its opposite. The main procedure of the algorithm is given below:

1. **algorithm** LIVE(**inputs :** E^+, E^-: training sets,
2. B: background theory, **outputs :** H : learned theory)
3. LearnDefinition($E^+, E^-, B; H_p$)
4. LearnDefinition($E^-, E^+, B; H_{\neg p}$)
5. Obtain H by:
6. transforming H_p, $H_{\neg p}$ into "non-deterministic" rules,
7. adding the clauses for the undefined case
8. **output** H

The algorithm calls a procedure LearnDefinition that, given a set of positive, a set of negative examples and a background knowledge, returns a definition for the positive concept, consisting of default rules, together with definitions for abnormality literals if any. The procedure LearnDefinition is called twice, once for the positive concept and once for the negative concept. When it is called for the negative concept, E^- is used as the positive training set and E^+ as the negative one.

LearnDefinition first calls a procedure Learn($E^+, E^-, B; H_p$) that learns a definition H_p for the target concept p. Learn consists of an ordinary ILP algorithm, either bottom-up or top-down, modified to adopt the SLX interpreter for testing the coverage of examples and to relax the consistency requirement of the solution. The procedure thus returns a theory that may cover some opposite examples. These opposite examples are then treated as exceptions, by adding a default literal to the inconsistent rules and adding proper facts for the abnormality predicate. In particular, for each rule $r = p(\vec{X}) \leftarrow Body(\vec{X})$ in H_p covering some negative examples, a new non-abnormality literal *not* $abnormal_r(\vec{X})$ is added to r and some facts for $abnormal_r(\vec{X})$ are added to the theory. Examples for $abnormal_r$ are obtained from examples for p by observing that, in order to cover an example $p(\vec{t})$ for p, the atom $abnormal_r(\vec{t})$ must be false. Therefore, facts for $abnormal_r$ are obtained from the set E_r^- of opposite examples covered by the rule.

1. **procedure** LearnDefinition(**inputs :** E^+: positive examples,

E^-: negative examples, B: background theory,
outputs : H: learned theory)
Learn($E^+, E^-, B; H_p$)
$H := H_p$
for each rule r in H_p **do**
    Find the sets E_r^+, E_r^- of positive and negative examples covered by r
    if E_r^- is not empty **then**
        Add the literal $not\_abnormal_r(\vec{X})$ to r
        Obtain $H_r = \{abnormal_r(\vec{t})\}$ from facts in E_r^- by
        transforming each $p(\vec{t}) \in E_r^-$ into $abnormal_r(\vec{t})$
        $H := H \cup H_r$
    endif
enfor
output H

Let us now discuss in more detail the algorithm that implements the Learn procedure. Depending on the generality of solution that we want to learn, different algorithms must be employed: a top-down algorithm for learning the MGS, a bottom-up algorithm for the LGS. In both cases, the algorithm must be such that, if a consistent solution cannot be found, it returns a theory that covers the least number of negative examples.

When learning with a top-down algorithm, the consistency necessity stopping criterion must be relaxed to allow clauses that are inconsistent with a small number of negative examples, e.g., by adopting one of the heuristic necessity stopping criteria proposed in ILP to handle noise, such as the encoding length restriction [43] of FOIL [43] or the significancy test of mFOIL [18]. In this way, we are able to learn definitions of concepts with exceptions: when a clause must be specialized too much in order to make it consistent, we prefer to transform it into a default rule and consider the covered negative examples as exceptions. The simplest criterion that can be adopted is to stop specializing the clause when no literal from the language bias can be added that reduces the coverage of negative examples.

When learning with a bottom-up algorithm, we can learn using positive examples only by using the *rlgg* operator: since the clause is not tested on negative examples, it may cover some of them. This approach is realized by using the system GOLEM, as in [25].

7. Implementation

In order to learn the most general solutions, a top-down ILP algorithm (cf. [31]) has been integrated with the procedure *SLX* for testing the coverage. The specialization loop of the top-down system consists of a beam search in the space of possible clauses. At each step of the loop, the system removes the best clause from the beam and generates refinements. They are then evaluated according to an accuracy heuristic function, and their refinements covering at least one positive example are added to the beam. The best clause found so far is also separately stored: this clause is compared with each refinement and is replaced if the refinement is better. The specialization loop stops when either the best clause in the beam is consistent or the beam becomes empty. Then the system returns the best clause found so far. The beam may become empty before a consistent clause is found and in this case the system will return an inconsistent clause.

In order to find least general solutions, the GOLEM [37] system is employed. The finite well-founded model is computed, through *SLX*, and it is transformed by replacing literals of the form $\neg A$ with new predicate symbols of the form neg_A. Then GOLEM is called with the computed model as background knowledge. The output of GOLEM is then parsed in order to extract the clauses generated by $rlgg$ before they are post-processed by dropping literals. Thus, the clauses that are extracted belong to the least general solution. In fact, they are obtained by randomly picking couples of examples, computing their $rlgg$ and choosing the consistent one that covers the biggest number of positive examples. This clause is further generalized by choosing randomly new positive examples and computing the $rlgg$ of the previously generated clause and each of the examples. The consistent generalization that covers more examples is chosen and further generalized until the clause starts covering some negative examples. An inverse model transformation is then applied to the rules thus obtained by substituting each literal of the form neg_A with the literal $\neg A$.

LIVE was implemented in *XSB* Prolog [45] and the code of the system can be found at :
`http://www-lia.deis.unibo.it/Software/LIVE/`.

8. Classification Accuracy

In this section, we compare the accuracy that can be obtained by means of a two-valued definition of the target concept with the one that can be obtained by means of a three-valued definition.

The accuracy of a two-valued definition over a set of testing examples is defined as

$$Accuracy_2 = \frac{\text{number of examples correctly classified by the theory}}{\text{total number of testing examples}}$$

The number of examples correctly classified is given by the number of positive examples covered by the learned theory plus the number of negative examples not covered by the theory. If Np is the number of positive examples covered by the learned definition, Nn the number of negative examples covered by the definition, $Nptot$ the total number of positive examples and $Nntot$ the total number of negative examples, then the accuracy is given by:

$$Accuracy_2 = \frac{Np + Nntot - Nn}{Nptot + Nntot}$$

When we learn in a three value setting a definition for a target concept and its opposite, we have to consider a different notion of accuracy. In this case, some atoms (positive or negative) in the testing set will be classified as undefined. Undefined atoms are covered by both the definition learned for the positive concept and that learned for the opposite one. Whichever is the right classification of the atom in the test set, it is erroneously classified in the learned three-valued theory, but not so erroneously as if it was covered by the opposite definition only. This explains the weight assigned to undefined atoms (i.e., 0.5) in the new, generalized, notion of accuracy:

$$Accuracy_3 = \frac{\text{number of examples correctly classified by the theory}}{\text{total number of testing examples}} +$$

$$+0.5 \times \frac{\text{number of examples classified as unknown}}{\text{total number of testing examples}}$$

In order to get a formula to calculate the accuracy, we first define a number of figures that are illustrated in figure 4:

- Npp is the number of positive examples covered by the positive definition only,
- Npn is the number of positive examples covered by the negative definition only,
- Npu is the number of positive examples covered by both definitions (classified as undefined),
- Nnn is the number of negative examples covered by the negative definition only,
- Nnp is the number of negative examples covered by the positive definition only,
- Nnu is the number of negative examples covered by both definitions (classified as undefined).

The accuracy for the three-valued case can thus be defined as follows:

$$Accuracy_3 = \frac{Npp + Nnn + 0.5 \times Npu + 0.5 \times Nnu}{Nptot + Nntot}$$

It is interesting to compare this notion of accuracy with that obtained by testing the theory in a two-valued way. In that case the accuracy would be given by:

$$Accuracy_2 = \frac{Npp + Npu + Nntot - Nnp - Nnu}{Nptot + Nntot}$$

We are interested in situations where the accuracy for the three-valued case is higher than the one for the two-valued case, i.e., those for which $Accuracy_3 > Accuracy_2$. By rewriting this inequation in terms of the figures above, we get:

$$Npp + 0.5 \times Npu + Nnn + 0.5 \times Nnu > Npp + Npu + Nntot - Nnp - Nnu$$

This inequation can be rewritten as:

$$Nntot - Nnn - Nnp < 1.5 \times Nnu - 0.5 \times Npu$$

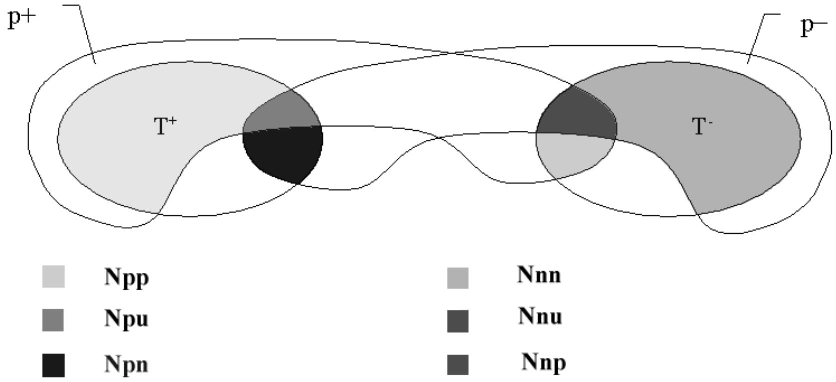

Figure 4. Sets of examples for evaluating the accuracy of a three-valued hypothesis

where the expression $Nntot - Nnn - Nnp$ represents the number of negative examples not covered by any definition (call it $Nn\_not\_covered$). Therefore, the accuracy that results from testing the theory in a three-valued way improves the two-valued one when most of the negative examples are covered by any of the two definitions, the number of negative examples on which there is contradiction is particularly high, and the number of positive examples on which there is contradiction is low.

When there is no overlap between the two definitions, and no undefinedness, the accuracy is the same.

9. Related Work

The adoption of negation in learning has been investigated by many authors. Many propositional learning systems learn a definition for both the concept and its opposite. For example, systems that learn decision trees, as c4.5 [42], or decision rules, as the AQ family of systems [33], are able to solve the problem of learning a definition for n classes, that generalizes the problem of learning a concept and its opposite. However, in most cases the definitions learned are assumed to cover the whole universe of discourse: no undefined classification is produced, any instance is always classified as belonging to one of the classes. Instead, we classify as undefined the instances for which the learned definitions do not give a unanimous response.

When learning multiple concepts, it may be the case that the descriptions learned are overlapping. We have considered this case as non-desirable: this is reasonable when learning a concept and its opposite but it may not be the case when learning more than two concepts (see [17]). As it has been pointed out by [34], in some cases it is useful to produce more than one classification for an instance: for example if a patient has two diseases, his symptoms should satisfy the descriptions of both diseases. A subject for future work will be to consider classes of paraconsistent logic programs where the overlap of definitions for p and $\neg p$ (and, in general, multiple concepts) is allowed.

The problems raised by negation and uncertainty in concept-learning, and Inductive Logic Programming in particular, were pointed out in some previous work (e.g., [4, 13, 10]). For concept learning, the use of the CWA for target predicates is no longer acceptable because it does not allow to distinguish between what is false and what is undefined. De Raedt and Bruynooghe [13] proposed to use a three-valued logic (later on formally defined in [10]) and an explicit definition of the negated concept in concept learning. This technique has been integrated within the CLINT system, an interactive concept-learner. In the resulting system, both a positive and a negative definition are learned for a concept (predicate) p, stating, respectively, the conditions under which p is true and those under which it is false. The definitions are learned so that they do not produce an inconsistency on the examples. Furthermore, CLINT does not produce inconsistencies also on unseen examples because of its constraint handling mechanism, since it would assert the constraint $p, not\ p \rightarrow false$, and take care that it is never violated. Distinctly from this system, we make sure that the two definitions do not produce inconsistency on unseen atoms by making learned rules non-deterministic. This way, we are able to learn definitions for exceptions to both concepts so that the information about contradiction is still available. Another contradistinction is that we cope with and employ simultaneously two kinds of negation, the explicit one, to state what is false, and the default (defeasible) one, to state what can be assumed false.

The system LELP (Learning Extended Logic Programs) [25] learns ELPs under answer-set semantics. LELP is able to learn non-deterministic default rules with a hierarchy of exceptions. Hierarchical learning of exceptions can be easily introduced in our system (see [27]). From the viewpoint of the learning problems that the two algorithms can solve, they are equivalent when the background is a stratified extended logic program, because then our and their semantics coincide. All the examples shown in [25] are stratified and therefore they can be learned by our algorithm and, viceversa, example in section 5 can be learned by LELP. However, when the background is a non-stratified extended logic program, the adoption of a well-founded semantics gives a number of advantages with respect to the answer-set semantics. For non-stratified background theories, answer-sets semantics does not enjoy the structural property of relevance [15], like our *WFSX* does, and so they cannot employ any top-down proof procedure. Furthermore, answer-set semantics is not cumulative [15], i.e., if you add a lemma then the semantics can change, and thus the improvement in efficiency given by tabling cannot be obtained. Moreover, by means of *WFSX*, we have introduced a method to choose one concept when the other is undefined which they cannot replicate because in the answer-set semantics one has to compute eventually all answer-sets to find out if a literal is undefined. The structure of the two algorithms is similar: LELP first generates candidate rules from a concept using an ordinary ILP framework. Then exceptions are identified (as covered examples of the opposite set) and rules specialized through negation as default and abnormality literals, which are then assumed to prevent the coverage of exceptions. These assumptions can be, in their turn, generalized to generate hierarchical default rules. One difference between us and [25] is in the level of generality of the definitions we can learn. LELP learns a definition for a concept only from positive examples of that concept and therefore it can only employ a bottom-up ILP technique and learn the LGS. Instead, we can choose whether to adopt a bottom-up or a top-down algorithm, and we can learn theories of different generality for different target concepts by integrating, in a declarative way, the learned definitions into a single ELP. Another difference consists in that LELP learns a definition only for the concept that has the highest number of examples in the training set. It learns both positive and negative concepts only when the number of positive examples is close to that of negative ones (in 60 %-40 % range), while we always learn both concepts.

Finally, many works have considered multi-strategy learners or multi-source learners. A multi-strategy learner combines learning strategies to produce effective hypotheses (see [26]). A multi-source learner implements an algorithm for integrating knowledge produced by the separate learners. Multi-strategy learning has been adopted, for instance, for the improvement of classification accuracy [17], and to equip an autonomous agent with capabilities to survive in an hostile environment [11].

Our approach considers two separate concept-based learners, in order to learn a definition for a concept and its opposite. Multiple (opposite) target concepts constitute part of the learned knowledge base, and each learning element is able to adopt a bottom-up or a top-down strategy in learning rules. This can be easily generalized to learn definitions for n disjoint classes of concepts or for multiple agent learning (see our [28]). Very often, the hypothesis can be more general than what is required. The second step of our approach, devoted to the application of strategies for eliminating learned contradictions, can be seen as a multi-source learner [26] or a meta-level one [6], where the learned definitions are combined to obtain a non-contradictory extended logic program. ELPs are used to specify combinations of strategies in a declarative way, and to recover, in the the process, the consistency of the learned theory.

10. Concluding Highlights

The two-valued setting that has been adopted in most work on ILP and Inductive Concept Learning in general is not sufficient in many cases where we need to represent real world data. This is for example the case of an agent that has to learn the effect of the actions it can perform on the domain by performing experiments. Such an agent needs to learn a definition for allowed actions, forbidden actions and actions with an unknown outcome, and therefore it needs to learn in a richer three-valued setting.

In order to achieve that in ILP, the class of extended logic programs under the well-founded semantics with explicit negation (*WFSX*) is adopted by us as the representation language. This language allows two kinds of negation, default negation plus a second form of negation called explicit, that is mustered in order to explicitly represent negative information. Adopting extended logic programs in ILP prosecutes the general trend in Machine Learning of extending the representation language in order to overcome the recognized limitations of existing systems.

The programs that are learned will contain a definition for the concept and its opposite, where the opposite concept is expressed by means of explicit negation. Standard ILP techniques can be adopted to separately learn the definitions for the concept and its opposite. Depending on the adopted technique, one can learn the most general or the least general definition.

The two definitions learned may overlap and the inconsistency is resolved in a different way for atoms in the training set and for unseen atoms: atoms in the training set are considered exceptions, while unseen atoms are considered unknown. The different behaviour is obtained by employing negation by default in the definitions: default abnormality literals are used in order to consider exceptions to rules, while non-deterministic rules are used in order to obtain an unknown value for unseen atoms. We have shown how the adoption of extended logic programs in ILP allows to tackle both learning in a three-valued setting and specify the combination of strategies in a declarative way, also coping with contradiction and exceptions in the process.

The system LIVE (Learning in a three-Valued Environment) has been developed to implement the above mentioned techniques. In particular, the system learns a definition for both the concept and its opposite and is able to identify exceptions and treat them through default negation. The system is parametric in the procedure used for learning each definition: it can adopt either a top-down algorithm, using beam-search and a heuristic necessity stopping criterion, or a bottom-up algorithm, that exploits the GOLEM system.

Notes

1. For definitions and foundations of LP, refer to [16]. For a recent state-of-the art of LP extensions for non-monotonic reasoning, refer to [2].
2. For the most advanced, incorporating more recent theoretical developments, see the XSB system at: `htpp://www.cs.sunysb.edu/~sbprolog/xsb-page.html`.
3. Notice that in the formula *not* $lover(H, L)$ variable H is universally quantified, whereas L is existentially quantified.
4. By non-contradictory program we mean a program which admits at least one $WFSX$ model.
5. The *undefined* predicate can be implemented through negation NOT under CWA ($NOT\ P$ means that P is false whereas *not* means that P is false or undefined), i.e., $undefined(P) \leftarrow NOT\ P, NOT(not\ P)$.

References

1. J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Proc. Int. Symp. on Logic Programming.* The MIT Press, 1994.
2. J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer-Verlag, 1996.

3. J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. "Classical" negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20:107–142, 1998.
4. M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive Logic Programming*, pages 145–161. Academic Press, 1992.
5. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
6. P.K. Chan and S.J. St9lfo. Meta-learning for multistrategy and parallel learning. In *Proceedings of the 2nd International Workshop on Multistrategy Learning*, pages 150–165, 1993.
7. C. V. Damásio, W. Nejdl, and L. M. Pereira. REVISE: An extended logic programming system for revising knowledge bases. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Knowledge Representation and Reasoning*, pages 607–618. Morgan Kaufmann, 1994.
8. C. V. Damásio and L. M. Pereira. Abduction on 3-valued extended logic programs. In V. W. Marek, A. Nerode, and M. Trusczynski, editors, *Logic Programming and Non-Monotonic Reasoning - Proc. of 3rd International Conference LPNMR'95*, volume 925 of *LNAI*, pages 29–42, Germany, 1997. Springer-Verlag.
9. C. V. Damásio and L. M. Pereira. A survey on paraconsistent semantics for extended logic programs. In D.M. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, pages 241–320. Kluwer Academic Publishers, 1998.
10. L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, 1992.
11. L. De Raedt, E. Bleken, V. Coget, C. Ghil, B. Swennen, and M. Bruynooghe. Learning to survive. In *Proceedings of the 2nd International Workshop on Multistrategy Learning*, pages 92–106, 1993.
12. L. De Raedt and M. Bruynooghe. Towards friendly concept-learners. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 849–856. Morgan Kaufmann, 1989.
13. L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept learning. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990.
14. L. De Raedt and M. Bruynooghe. Interactive concept learning and constructive induction by analogy. *Machine Learning*, 8(2):107–150, 1992.
15. J. Dix. A classification-theory of semantics of normal logic programs: I. & II. *Fundamenta Informaticae*, XXII(3):227–255 and 257–288, 1995.
16. J. Dix, L. M. Pereira, and T. Przymusinski. Prolegomena to logic programming and non-monotonic reasoning. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming - Selected papers from NMELP'96*, number 1216 in LNAI, pages 1–36, Germany, 1997. Springer-Verlag.
17. M. Drobnic and M. Gams. Multistrategy learning: An analytical approach. In *Proceedings of the 2nd International Workshop on Multistrategy Learning*, pages 31–41, 1993.
18. S. Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, 1991.
19. F. Esposito, S. Ferilli, E. Lamma, P. Mello, M. Milano, F. Riguzzi, and G. Semeraro. Cooperation of abduction and induction in logic programming. In P. A. Flach and A. C. Kakas, editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer, 1998. Submitted for publication.
20. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
21. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming ICLP90*, pages 579–597. The MIT Press, 1990.
22. D. Gordon and D. Perlis. Explicitly biased generalization. *Computational Intelligence*, 5(2):67–81, 1989.
23. R. Greiner, A. J. Grove, and D. Roth. Learning active classifiers. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML96)*, 1996.

24. K. Inoue. Learning abductive and nonmonotonic logic programs. In P. A. Flach and A. C. Kakas, editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer, 1998. Submitted for publication.
25. K. Inoue and Y. Kudoh. Learning extended logic programs. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 176–181. Morgan Kaufmann, 1997.
26. W.T. Jenkins. Intelog: A framework for multistrategy learning. In *Proceedings of the 2nd International Workshop on Multistrategy Learning*, pages 58–65, 1993.
27. E. Lamma, F. Riguzzi, and L. M. Pereira. Learning in a three-valued setting. In *Proceedings of the Fourth International Workshop on Multistrategy Learning*, 1988.
28. E. Lamma, F. Riguzzi, and L. M. Pereira. Agents learning in a three-valued setting. Technical report, DEIS - University of Bologna, 1999.
29. E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in combined learning via logic programs. Technical report, DEIS - University of Bologna, 1999.
30. S. Lapointe and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Workshop on Machine Learning*, pages 273–281. Morgan Kaufmann, 1992.
31. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.
32. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Collected Papers from Workshop on Logic Programming and Knowledge Representation LPKR'97*, number 1471 in LNAI. Springer-Verlag, 1998.
33. R. Michalski. Discovery classification rules using variable-valued logic system VL1. In *Proceedings of the Third International Conference on Artificial Intelligence*, pages 162–172. Stanford University, 1973.
34. R. Michalski. A theory and methodology of inductive learning. In R. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 1, pages 83–134. Springer-Verlag, 1984.
35. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
36. S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming*, pages 261–280. Academic Press, 1992.
37. S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.
38. M. J. Pazzani, C. Merz, P. Murphy, K. Ali, T. Hume, and C. Brunk. Reducing misclassification costs. In *Proceedings of the Eleventh International Conference on Machine Learning (ML94)*, pages 217–225, 1994.
39. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligenece ECAI92*, pages 102–106. John Wiley and Sons, 1992.
40. G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
41. F. J. Provost and T. Fawcett. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distribution. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD97)*. AAAI Press, 1997.
42. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.
43. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
44. R. Reiter. On closed-word data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.
45. K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. *The XSB Programmer's Manual Version 1.7.1*, 1997.
46. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Appendix : Definition of WFSX

The definition of *WFSX* that follows is taken from [1] and is based on the alternating fix points of Gelfond-Lifschitz Γ-like operators.

Definition 2. [The Γ-operator] Let P be an extended logic program and let I be an interpretation of P. $\Gamma_P(I)$ is the program obtained from P by performing in the sequence the following four operations:

- Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$.
- Remove from P all rules containing in the body an objective literal L such that $\neg L \in I$.
- Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$.
- Replace all the remaining default literals by proposition **u**.

In order to impose the coherence requirement, we need the following definition.

Definition 3. [Seminormal Version of a Program] The seminormal version of a program P is the program P_s obtained from P by adding to the (possibly empty) *Body* of each rule $L \leftarrow Body$ the default literal $not\neg L$, where $\neg L$ is the complement of L with respect to explicit negation.

In the following, we will use the following abbreviations: $\Gamma(S)$ for $\Gamma_P(S)$ and $\Gamma_s(S)$ for $\Gamma_{P_s}(S)$.

Definition 4. [Partial Stable Model] An interpretation $T \cup not\ F$ is called a *partial stable model* of P iff $T = \Gamma\Gamma_s T$ and $F = H^E(P) - \Gamma_s T$.

Partial stable models are an extension of *stable models* [20] for extended logic programs and a three-valued semantics. Not all programs have a partial stable model (e.g., $P = \{a, \neg a\}$) and programs without a partial stable model are called *contradictory*.

THEOREM 1 (*WFSX* SEMANTICS) *Every non-contradictory program P has a least (with respect to $\subseteq$) partial stable model, the well-founded model of P denoted by $WFM(P)$. To obtain an iterative "bottom-up" definition for $WFM(P)$ we define the following transfinite sequence $\{I_\alpha\}$:*

$$I_0 = \{\}; \quad I_{\alpha+1} = \Gamma\Gamma_S I_\alpha; \quad I_\delta = \bigcup\{I_\alpha | \alpha < \delta\}$$

where δ is a limit ordinal. There exists a smallest ordinal λ for the sequence above, such that I_λ is the smallest fix point of $\Gamma\Gamma_S$. *Then, $WFM(P) = I_\lambda \cup not\ (H^E(P) - \Gamma_S I_\lambda)$.*